24장. 클로저

1. **클로저(closure)**

   : 함수를 일급 객체로 취급하는 함수형 프로그래밍 언어에서 사용되는 중요한 특성

2. [예제 24-01]

```
const x = 1;

function outerFunc() {
    const x = 10;

    function innerFunc() {   // 중첩 함수
        console.log(x);  // 10
    }

    innerFunc();
}

outerFunc();
```

   - innerFunc 내부에서 자신을 포함하고 있는 외부 함수 outerFunc의 x변수에 접근할 수 있다.

3. [예제 24-02]

```
const x = 1;

function outerFunc() {
    const x = 10;
    innerFunc();
}

function innerFunc() {
    console.log(x);  // 1
```

```
}

outerFunc();
```

- 이 같은 현상이 발생하는 이유는,

  자바스크립트가 렉시컬 스코프를 따르는 프로그래밍 언어이기 때문이다.

24.1 렉시컬 스코프

1. 자바스크립트 엔진은 함수를 어디서 호출했는지가 아니라,

   함수를 어디에 정의했는지에 따라 상위 스코프를 결정한다.

   이를, 렉시컬 스코프(정적 스코프)라 함.

2. [예제 24-03]

```
const x = 1;

function foo() {
    const x = 10;
    bar();
}

function bar() {
    console.log(x);
}

foo();  // ?
bar();  // ?
```

- foo(), bar() 모두 전역에서 정의된 함수 (이들의 상위 스코프는 전역)
- 함수의 상위 스코프는 함수를 어디서 정의했느냐에 따라 결정.

  즉, 함수의 상위 스코프는 함수를 정의한 위치에 의해 정적으로 결정되고 변화 X
- "함수의 상위 스코프를 결정한다"

= 렉시컬 환경의 외부 렉시컬 환경에 대한 참조에 저장할 참조값을 결정한다

- **"외부 렉시컬 환경에 대한 참조"에 저장할 참조값, 즉 상위 스코프에 대한 참조는,**
  함수 정의가 평가되는 시점에 함수가 정의된 환경(위치)에 의해 결정
  = 렉시컬 스코프

24.2 함수 객체의 내부 슬롯 [[Environment]]

1. 함수는 자신이 호출되는 환경과는 상관없이 자신이 정의된 환경,
   즉 상위 스코프를 기억해야 함.
   ➡️ **이를 위해 함수는 자신의 내부 슬롯 [[Environment]]에 자신이 정의된 환경,**
   즉 상위 스코프의 참조를 저장한다.

2. **함수 객체의 내부 슬롯 내부 슬롯 [[Environment]]에 저장된 현재 실행 중인 실행 컨텍스트의**
   렉시컬 환경의 참조가 상위 스코프. 또한 자신이 호출됐을 때 생성될 함수 렉시컬 환경의 "외부
   렉시컬 환경에 대한 참조"에 저장될 참조값이다. 함수 객체는 내부 슬롯 [[Environment]]에 저장
   한 렉시컬 환경의 참조, 즉 상위 스코프를 자신이 존재하는 한 기억함.

24.3 클로저와 렉시컬 환경

1. [예제 24-05]

```
const x = 1;

// ①
function outer() {
    const x = 10;
    const inner = function () { console.log(x); };  // ②
```

```
    return inner;
}

// outer 함수를 호출하면 중첩 함수 inner를 반환
// 그리고 outer 함수의 실행 컨텍스트는 실행 컨텍스트 스택에서 팝되어 ㅈ
const innerFunc = outer();  // ③
innerFunc();   // ④
```

- outer 함수와 함께 생명 주기를 마감한 지역 변수 x가 다시 부활이라도 한 듯이 동작.

➡️ **외부 함수보다 중첩 함수가 더 오래 유지되는 경우 중첩함수는 이미 생명 주기가 종료한**

외부 함수의 변수를 참조할 수 있다. 이러한 중첩 함수를 클로저(closure)라 부른다.

2. **outer 함수의 실행 컨텍스트는 실행 컨텍스트 스택에서 제거되지만,**

   outer 함수의 렉시컬 환경까지 소멸하는 것은 아니다.

3. [예제 24-06]

```
function foo() {
    const x = 1;
    const y = 2;

    // 일반적으로 클로저라고 하지 않음.
    function bar() {
        const z = 3;

        debugger;
        // 상위 스코프의 식별자를 참조하지 X
        console.log(z);
    }

    return bar;
}
```

```
const bar = foo();
bar();
```

- 중첩 함수 bar는 외부 함수 foo보다 더 오래 유지되지만 상위 스코프의 어떤 식별자도 참조 X

  → 따라서 bar 함수는 클로저라고 할 수 없다.

4. [예제 24-07]

```
function foo() {    // 외부 함수
    const x = 1;
    const y = 2;

    // bar 함수는 클로저였지만 곧바로 소멸.
    // 이러한 함수는 일반적으로 클로저라고 하지 않음.
    function bar() {    // 중첩 함수
        debugger;
        // 상위 스코프의 식별자를 참조한다.
        console.log(x);
    }
    bar();
}

foo();
```

- 중첩 함수 bar는 상위 스코프 식별자를 참조하고 있으므로 클로저다.

  하지만 외부 함수 foo의 외부로 중첩 함수 bar가 반환되지 X

  즉, 외부 함수 foo보다 중첩 함수 bar의 생명 주기가 짧다 = 클로저가 아니다.

5. [예제 24-08]

```
function foo() {    // 외부 함수
    const x = 1;  // 상위 스코프의 변수, x를 자유 변수(free variabl
    const y = 2;

    // 클로저
```

```
    // 중첩 함수 bar는 외부 함수보다 더 오래 유지되며, 상위 스코프의 식별
    function bar() {    // 중첩 함수
        debugger;
        console.log(x);
    }
    return bar();
}

const bar = foo();
bar();
```

결론. 클로저의 조건

> **1. 중첩 함수가 상위 스코프의 식별자를 참조하고 있고,**

> **2. 중첩 함수가 외부 함수보다 더 오래 유지되는 경우에 한정**

24.4 클로저의 활용

1. 클로저는 **상태(state)를 안전하게 변경 & 유지하기 위해 사용**한다.

= 상태가 의도치 않게 변경되지 않도록 **상태를 안전하게 은닉하고 특정함수에게만 상태 변경 허용**

2. [예제 24-09]

```
// 카운트 상태 변수
let num = 0;

// 카운트 상태 변경 함수
const increase = function() {
    return ++num;   // 카운트 1만큼 증가
};

console.log(increase());  // 1
```

```
console.log(increase());  // 2
console.log(increase());  // 3
```

오류 발생시킬 가능성 있는 좋지 않은 코드

< ❗ 위 예제가 바르게 동작하는 **전제 조건** >

1. *카운트 상태(num 변수 값)는 increase 함수 호출되기 전까지 변경되지 않고 유지되어야 함.*
2. *이를 위해 카운트 상태(num 변수의 값)는 increase 함수만이 변경할 수 있어야 함.*

---

3. [예제 24-10]

```
// 카운트 상태 변경 함수
const increase = function() {
    // 카운트 상태 변수
    let num = 0;

    return ++num;   // 카운트 1만큼 증가
};

// 이전 상태를 유지하지 못한다.
console.log(increase());  // 1
console.log(increase());  // 1
console.log(increase());  // 1
```

- 전역 변수 num을, 함수 increase의 지역 변수로 변경하여 의도치 않은 상태 변경은 방지했다.
- 하지만 함수가 호출될 때마다 지역 변수 num은 다시 선언되고 0으로 초기화된다.

  = 상태가 변경되기 이전 상태를 유지하지 못한다.

4. [예제 24-11]

```
// 카운트 상태 변경 함수
const increase = ( function() {
    // 카운트 상태 변수
    let num = 0;

    // 클로저
    return function () {
        return ++num;   // 카운트 1만큼 증가
    };
}());

console.log(increase());  // 1
console.log(increase());  // 2
console.log(increase());  // 3
```

▶ **클로저는 상태가 의도치 않게 변경되지 않도록 상태를 안전하게 은닉하고,**

특정함수에게만 상태 변경을 허용하여 상태를 안전하게 변경 & 유지하기 위해 사용

5. [예제 24-12]

```
const counter = (function () {  // 즉시 실행 함수
  // 카운트 상태 변수
  let num = 0;

  // 클로저인 메서드를 갖는 객체를 반환.
  // 객체 리터럴은 스코프를 만들지 않는다.
  // 따라서 아래 메서드들의 상위 스코프는 즉시 실행 함수의 렉시컬 환경(
  return {
    // num: 0,  // 프로퍼티는 public하므로 은닉되지 않는다.
    increase() {
      return ++num;
    },
    decrease() {
      return num > 0 ? --num : 0;
    }
  };
}());
```

```
console.log(counter.increase()); // 1
console.log(counter.increase()); // 2

console.log(counter.decrease()); // 1
console.log(counter.decrease()); // 0
```

위 예제를 생성자 함수로 표현하면,

[예제 24-13]

```
const Counter = (function () {
  // ① 카운트 상태 변수
  let num = 0;

  function Counter() {
    // this.num = 0; // ② 프로퍼티는 public하므로 은닉되지 않는
  }

  Counter.prototype.increase = function () {
    return ++num;
  };

  Counter.prototype.decrease = function () {
    return num > 0 ? --num : 0;
  };

  return Counter;
}());

const counter = new Counter();

console.log(counter.increase()); // 1
console.log(counter.increase()); // 2

console.log(counter.decrease()); // 1
console.log(counter.decrease()); // 0
```

6. [예제 24-14]

```
// 함수를 인수로 전달받고 함수를 반환하는 고차 함수
// 이 함수는 카운트 상태를 유지하기 위한 자유 변수 counter를 기억하는 클로
function makeCounter(aux) {
  // 카운트 상태를 유지하기 위한 자유 변수
  let counter = 0;

  // 클로저를 반환
  return function () {
    // 인수로 전달 받은 보조 함수에 상태 변경을 위임한다.
    counter = aux(counter);
    return counter;
  };
}

// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// 함수로 함수를 생성한다.
// makeCounter 함수는 보조 함수를 인수로 전달받아 함수를 반환한다
const increaser = makeCounter(increase); // ①
console.log(increaser()); // 1
console.log(increaser()); // 2

// increaser 함수와는 별개의 독립된 렉시컬 환경을 갖기 때문에 카운터 상태
const decreaser = makeCounter(decrease); // ②
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

- makeCounter 함수를 호출해 함수를 반환할 때 반환된 함수는,
  자신만의 독립된 렉시컬 환경을 갖는다.

7. [예제 24-15]

```
// 함수를 반환하는 고차 함수
// 이 함수는 카운트 상태를 유지하기 위한 자유 변수 counter를 기억하는 클로
const counter = (function () {
  // 카운트 상태를 유지하기 위한 자유 변수
  let counter = 0;

  // 함수를 인수로 전달받는 클로저를 반환
  return function (aux) {
    // 인수로 전달 받은 보조 함수에 상태 변경을 위임한다.
    counter = aux(counter);
    return counter;
  };
}());

// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// 보조 함수를 전달하여 호출
console.log(counter(increase)); // 1
console.log(counter(increase)); // 2

// 자유 변수를 공유한다.
console.log(counter(decrease)); // 1
console.log(counter(decrease)); // 0
```

- makeCounter 함수를 두 번 호출하지 않음.

24.5 캡슐화와 정보 은닉

1. **캡슐화(encapsulation)**

   : 객체의 상태를 나타내는 프로퍼티, 이를 참조 & 조작할 수 있는 메서드를 하나로 묶는 것

2. 캡슐화는 객체의 특정 프로퍼티나 메서드를 감출 목적으로 사용하기도 함.

   = **정보 은닉** (information hiding)

3. 대부분 객체지향 프로그래밍 언어는 public, private, protected 같은 접근 제한자가 있음.

   자바스크립트는 따로 제공 X, 즉 객체의 모든 프로퍼티, 메서드는 기본적으로 public

4. [예제 24-16]

```
function Person(name, age) {
  this.name = name; // public
  let _age = age;   // private

  // 인스턴스 메서드
  this.sayHi = function () {
    console.log(`Hi! My name is ${this.name}. I am ${_age}.`)
  };
}

const me = new Person('Lee', 20);
me.sayHi(); // Hi! My name is Lee. I am 20.
console.log(me.name); // Lee
console.log(me._age); // undefined

const you = new Person('Kim', 30);
```

```
you.sayHi(); // Hi! My name is Kim. I am 30.
console.log(you.name); // Kim
console.log(you._age); // undefined
```

- _age 변수는 private하다.
- sayHi 메서드는 Person 객체가 생성될 때마다 중복 생성된다.

5. [예제 24-17]

```
function Person(name, age) {
  this.name = name; // public
  let _age = age;   // private
}

// 프로토타입 메서드
Person.prototype.sayHi = function () {
  // Person 생성자 함수의 지역 변수 _age를 참조할 수 없다.
  console.log(`Hi! My name is ${this.name}. I am ${_age}.`);
};
```

⇒ 즉시 실행 함수 사용하여 생성자 함수와 메서드를 하나의 함수 내로 모아 보자.

[예제 24-18]

```
const Person = (function () {
  let _age = 0; // private

  // 생성자 함수
  function Person(name, age) {
    this.name = name; // public
    _age = age;
  }

  // 프로토타입 메서드
  Person.prototype.sayHi = function () {
    console.log(`Hi! My name is ${this.name}. I am ${_age}.`)
```

```
  };

  // 생성자 함수를 반환
  return Person;
}());

const me = new Person('Lee', 20);
me.sayHi(); // Hi! My name is Lee. I am 20.
console.log(me.name); // Lee
console.log(me._age); // undefined

const you = new Person('Kim', 30);
you.sayHi(); // Hi! My name is Kim. I am 30.
console.log(you.name); // Kim
console.log(you._age); // undefined
```

- 위 코드도 완벽하지는 X
- Person 생성자 함수가 여러 개의 인스턴스를 생성할 경우, _age 변수의 상태가 유지되지 X

[예제 24-19]

```
const me = new Person('Lee', 20);
me.sayHi(); // Hi! My name is Lee. I am 20.

const you = new Person('Kim', 30);
you.sayHi(); // Hi! My name is Kim. I am 30.

// _age 변수 값이 변경된다!
me.sayHi(); // Hi! My name is Lee. I am 30.
```

- Person.prototype.sayHi 메서드가 단 한 번 생성되는 클로저이기 때문에 발생하는 현상.
- Person.prototype.sayHi 메서드의 상위 스코프는 어떤 인스턴스로 호출하더라도 하나의 동일한 상위 스코프를 사용하게 된다.

→ 이러한 이유로 Person 생성자 함수가 여러 개의 인스턴스를 생성할 경우,

_age 변수의 상태가 유지되지 않는다.

결론

➡️ 자바스크립트는 정보 은닉을 완전하게 지원하지 않는다.

- ES6의 Symbol, WeakMap으로 private한 프로퍼티를 흉내 내기도 했으나 근본적인 해결책은 X

24.6 자주 발생하는 실수

1. [예제 24-20]

```
var funcs = [];

for (var i = 0; i < 3; i++) {
  funcs[i] = function () { return i; }; // ①
}

for (var j = 0; j < funcs.length; j++) {
  console.log(funcs[j]()); // ②
}
```

- 결과가 0, 1, 2가 나오길 기대했지만 그렇지 않다.
- var 키워드로 선언한 i 변수는 함수 레벨 스코프를 갖기 때문에 전역 변수다.
- 전역 변수 i에는 0, 1, 2, 3이 순차적으로 할당되어 결과는 i의 값 3이 출력된다.

2. [예제 24-21]

```
var funcs = [];

for (var i = 0; i < 3; i++){
  funcs[i] = (function (id) { // ①
```

```
    return function () {
      return id;
    };
  }(i));
}

for (var j = 0; j < funcs.length; j++) {
  console.log(funcs[j]());
}
```

- for문 변수 선언문에서 let 키워드로 선언한 변수 사용하면 for문의 코드 블록이 반복 실행될 때

  마다 for문 코드 블록의 **새로운 렉시컬 환경이 생성**된다.

3. 이처럼 let이나 const 키워드를 사용하는 반복문은 코드 블록이 반복 실행될 때마다,

   새로운 렉시컬 환경을 생성하여 반복할 당시의 상태를 **마치 스냅샷을 찍는 것처럼 저장한다.**

4. 또 다른 방법으로는 고차 함수를 사용하는 방법이 있다.